Quiz/Homework 4

The questions are designed by the TAs or me. For information on how to contact the TAs, please refer to the course outline.

Due Date: **From** April **1st** to April **5th** during your labs sessions.

Instructions

- Please ensure that you complete all questions and present your answers to your TAs during **one** of your **assigned** weekly lab sessions between Due Date. TAs may ask you questions and request you to work with your codes or present it. TAs will provide feedback and your grade right away. Grades will also be released on Avenue a few days later.
- The purpose of these Quiz/Homework assignments is not solely to test your programming skills. Instead, they are designed to provide you with simple and basic examples to better prepare you for your assignments. Consequently, you can work on the questions outside of the lab sessions. You may ask questions from your TAs at any point over Teams or during the lab sessions.
- Please attend only the labs for which you are enrolled to prevent classes from becoming overcrowded.
- You don't need to submit anything. Just keep your files organized to be presentable if the TAs asked you to work with your code.

1. **Dynamic Memory Allocation (Pedram, 2 points)**

   Let's go back to **Assignment 3 - Scientific Programming: Operations on Matrices** and make it a little bit more perfect. This time you can make any changes you want in the format even the file `functions.h`. **My** `functions.h` looks like:

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

// Function prototypes
void generateMatrix(double **matrix, int rows, int cols);
void printMatrix(double **matrix, int rows, int cols);
void addMatrices(int N1, int M1, double **A, int N2, int M2,
    double **B, double **result);
```

```
void subtractMatrices(int N1, int M1, double **A, int N2,
    int M2, double **B, double **result);
void multiplyMatrices(int N1, int M1, double **A, int N2,
    int M2, double **B, double **result);
void solveLinearSystem(int N1, int M1, double **A, int N2,
    int M2, double **B, double **x);

void deallocateMatrix(double **matrix, int rows);

#endif /* FUNCTIONS_H */
```

You **can** change this file if you want or need to.

Use Dynamic Memory Allocation to randomly create 2D arrays A and B with double-precision random numbers from -10 to 10. Since I used `malloc` to allocate memory on Heap, I implemented a function to de-allocate the memory called `void deallocateMatrix()`.

Then, for matrix A with $N1$ rows and $M1$ columns and B with $N2$ rows and $M2$ columns, your program should be able to compute $A + B$, $A - B$, $A \times B$, and solve $Ax = B$ for x. **Technically, everything is the same as Assignment 3, plus having dynamic memory allocation where ever it is necessary**.

Make sure the results are correct for small size of matrices. Then, run your code for large matrices like:

1. `./math 20000 20000 20000 20000 add`
2. `./math 20000 20000 20000 20000 subtract`
3. `./math 1500 1500 1500 1500 multiply`
4. `./math 3000 3000 3000 1 solve`

If you get a `segmentation fault`, probably the memory allocation has an issue. Write and keep the CPU time taken for all the above runs in a paper or somewhere (You don't need to submit anything just show it to your TA if necessary). Keep the codes in files `functions.h`, `functions.c`, and `math_matrix.c` in case your TA wants to take a look.

Let's dig into it! You can read this part if you are interested in this topic. This is an advanced topic, and TAs will not ask you questions about it.
Probably you have noticed that if `N1=M1=20000`, then to save the matrix A on memory we need:

$$20000 \times 20000 \times 64 \text{ bits } \textbf{OR } 2.98 \text{ GBs}$$

Let's say B has the same size and if the operation is `add`, then the size of C will be the same. It means we need at least 3×2.98 GBs memory to perform the computation. So what happens to very large systems? For example numerical weather prediction models, solving systems of equations ($Ax = B$) representing atmospheric dynamics, moisture, and energy transfer. These models typically involve grids with millions of points, resulting in systems of equations with millions or even billions of unknowns. This means **only to save matrix** A with dimensions of 1 billion by 1 billion, it would take **7.45e+09 GBs** of memory.
Beside, Weather Forecasting, the linear solvers are used in many other fields like Structural Engineering, Image Processing, Optimization Problems, and Machine Learning. One way to overcome this issue is enhancing or adopting the algorithm to use memory and other resources efficiently. For example, if the matrix is Sparse, to avoid wasting the memory, we must save matrices in Coordinate Storage, Compressed Sparse Column, and Compressed Sparse Row formats.
Solving $Ax = B$ for x when A is sparse needs Sparse Linear Solvers. This is was just an introduction to this field, and you can find more details about it on this YouTube playlist.

2. **Memory Leak (Pedram, 2 points)**

In programming, particularly in languages like C, a memory leak occurs when a program allocates memory from the system (for example, using functions like `malloc()` or `calloc()`), but fails to de-allocate or free that memory when it is no longer needed. As a result, the memory remains allocated even though the program no longer uses it, which can lead to inefficient memory usage and, in extreme cases, can cause the program to run out of memory.

Memory leaks are a common issue in C programming due to the manual memory management nature of the language. Unlike languages with automatic garbage collection, such as Java or Python, C requires the programmer to explicitly allocate and de-allocate memory.

Valgrind is a powerful tool for detecting memory management and thread-related errors in programs, including memory leaks, memory corruption, and undefined behavior. It works on various Unix-like operating systems, including Linux and macOS.

Use Valgrind (ask **ChatGPT** how to use it) to find if there is any memory leak in codes for the **previous** question. If I don't de-allocate the memory for matrix `A` and `B` while I am executing the command `valgrind --leak-check=full ./math 5 4 5 4 subtract print`, I get the following information in the terminal:

```
<few lines of prints, and then:>
==26319== HEAP SUMMARY:
==26319==     in use at exit: 400 bytes in 12 blocks
==26319==   total heap usage: 19 allocs, 7 frees, 1,624 bytes allocated
==26319==
==26319== 200 (40 direct, 160 indirect) bytes in 1 blocks are definitely lost in loss record 3 of 4
==26319==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==26319==    by 0x10936A: main (in /home/pedram/COMPSCI1XC3/homework_quiz/Quiz4/math)
==26319==
==26319== 200 (40 direct, 160 indirect) bytes in 1 blocks are definitely lost in loss record 4 of 4
==26319==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==26319==    by 0x109439: main (in /home/pedram/COMPSCI1XC3/homework_quiz/Quiz4/math)
==26319==
==26319== LEAK SUMMARY:
==26319==    definitely lost: 80 bytes in 2 blocks
==26319==    indirectly lost: 320 bytes in 10 blocks
==26319==      possibly lost: 0 bytes in 0 blocks
==26319==    still reachable: 0 bytes in 0 blocks
==26319==         suppressed: 0 bytes in 0 blocks
==26319==
==26319== For lists of detected and suppressed errors, rerun with: -s
==26319== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

After fixing the issue I have:

```
<few lines of prints, and then:>
==26651== HEAP SUMMARY:
==26651==     in use at exit: 0 bytes in 0 blocks
==26651==   total heap usage: 19 allocs, 19 frees, 1,624 bytes allocated
==26651==
==26651== All heap blocks were freed -- no leaks are possible
==26651==
==26651== For lists of detected and suppressed errors, rerun with: -s
==26651== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Fix all the issues, then show the error free Valgrind output in the terminal to your TA.

3. **Binary Tree in C (Pedram, 2 points)**

Study about binary tree. To create a binary tree we need a `struct` with three members. `int value` can hold the number, `left` and `right` are also pointers to another `Node`. The nodes will connected on Heap part of memory using these pointers. Essentially, the following `struct` can represent each node in a binary tree:

```
struct Node {
 int value;
 struct Node* left;
 struct Node* right;
};
```

a) (1/2 point)You can use the file `quiz4_q3.c` and complete the code by writing the function `createNode()` to create the binary tree `root`. We want to create the following binary trees:

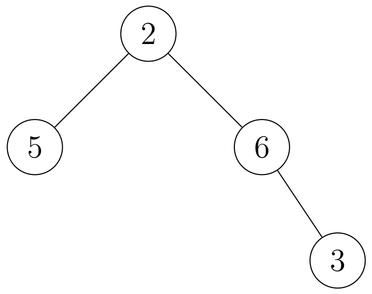

Figure 1: `root1`.

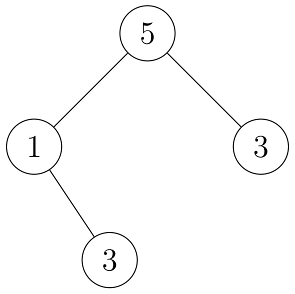

Figure 2: `root2`.

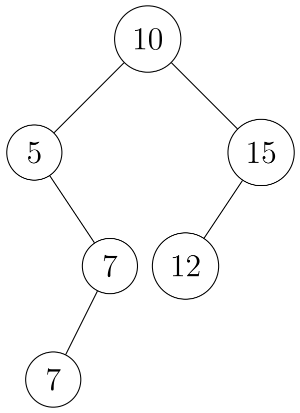

Figure 3: `root3`.

For example, to create `root1`, we must use:

```
    struct Node *root1 = createNode(2,NULL,NULL);
    root1->left = createNode(5,NULL,NULL);
    root1->right = createNode(6,NULL,NULL);
    root1->right->right = createNode(3,NULL,NULL);
```

Function `inorderTraversal()` will print the traversal graph in the terminal. Show the printed results in the terminal to you TA.

b) (1/2 point) Compile `quiz4_q3.c` and use **Valgrind** to check for memory leak. Every time you create a tree like `root1`, you must de-allocate the memory. Complete the code by writing a function to de-allocate the memory by passing `root` to the function.